A Toolkit for Developing Multi-User, Distributed Virtual Environments

Christopher. F. Codella, Reza Jalili, Lawrence Koved, J. Bryan Lewis
IBM Thomas J. Watson Research Center,
P.O. Box 704,
Yorktown Heights, New York 10598

Abstract

This paper reviews the design and operation of the Virtual Reality Distributed Environment and Construction Kit (VR-DECK) toolkit developed at IBM Research. Ease of use was emphasized both for the application builder and the toolkit extender. The system supports distributed processing, the building of multi-user shared environments, as well as a variety of specialized I/O devices such as gloves, 3-D position sensors, sound generation, speech recognition, and 3-D graphics displays, under an open and extensible architecture. Virtual world environments are created using a mixed object-oriented and event based paradigm for defining system behavior. Basic units, called modules, represent entities in the world such as objects, operations, functions or users. Modules communicate with each other by producing and consuming events, and are defined at a high-level using rules written in C++ that determine how events are handled. The toolkit, designed to be run on one or more workstations, includes a development environment consisting of C++ class libraries for module construction, interprocess communication, device support, and hierarchical object-oriented graphics. The run-time environment includes an X Window System™ based control panel for dynamically constructing worlds from a collection of modules, allocating processes among hosts, editing modules, and monitoring operation, and a library of ready to use modules for various devices and common operations.

Introduction

The term ***virtual world*** (a.k.a. virtual reality, virtual environment) generally refers to a human-computer interface allowing a user to experience a computer generated environment that is interactive and three dimensional. The virtual world and its objects can be represented by three-dimensional stereoscopic images and sounds. They are directly manipulated by a person through hand and body movements, and spoken words. Objects in the world may be linked to collections of data or to concurrently running simulators [1-4]. The technology brings together diverse elements including networked systems (possibly heterogeneous), specialized input and output devices, 3-D graphics, hand or body gesture and speech recognition, data visualization, distributed processing, simulation and the possibility of more than one user sharing the environment [4, 5].

In designing an architecture for constructing virtual worlds, we believe several key features are necessary in order to provide a reasonable balance between the opposing objectives of providing ease of use and building what is likely a very complex system. One must provide a framework within which the developer can see beyond this inherent complexity and concentrate creative energies at a

0-7803-1363-1/93 $3.00 © 1993 IEEE

high level on the construction of the world, its objects and their behavior. The system should be easy to use, both for the world (application) designer and the programmer who wishes to add extensions, such as new devices. Object oriented design, now a well established method for dealing with complexity, should be fully supported [6].

An application development environment should be modular and dynamic. By modular we mean that commonly used objects and functions should be built for reuse, providing an ever growing library of modules from which designers may choose when creating new applications. Furthermore, the modules should be self sufficient and capable of operating independently as well as in conjunction with other modules. In this way the construction of a virtual world is dynamic because experimentation is possible by substituting modules while the system is running. It is possible for the designer to build in stages, piece by piece, while observing its overall operation. The world functions fully or partially depending on whether or not all modules are present.

Finally, an application should be scalable over a wide range of system topologies. A world consructed on a single workstation should be transportable to a networked collection of processors without changing code or recompiling. It should be possible to make use of whatever resources are available with only minor modifications, if any.

Many of these objectives are shared by other toolkits (e.g. [7-9]).

Components: Modules, Events and Rules

In previous work [4, 5], we have described an architecture for virtual worlds consisting of a collection of device servers and other processes, communicating via asynchronous events processed through a central dialog manager or UIMS. Our current work takes this design a step further by distributing the role of the dialog manager among a collection of peer processes called modules. Each module is defined at the highest level by a set of rules governing the processing of events.

Modules can be thought of as the building blocks of a virtual world. They usually represent discrete objects or collections of objects, specific services (like I/O devices), or functions which can be encapsulated (like simulators). Intra- and inter-module communication occurs via events in a way analagous to a windowing system's use of events. Events are defined as discrete occurrances broadcast to other modules. In addition to an identifier or name, an event carries a strongly typed data object (possibly ***void***). The external view of a module is completely defined by the events it produces and consumes.

A module, at its highest level, is a collection of rules to handle the processing of events generated by the module, other modules, or external sources such as I/O devices and simulators. Extensive run-time support transparently handles event queuing, matching of events to rules that process the events, inter-module event and data transport, networking support, I/O device support and interfaces to the X Window System™. This run-time support, contained within C++ class libraries, is designed to relieve the world designer from low-level systems issues. It also provides the toolkit extender with a rich set of facilities upon which to enhance the base system.

Figure 1. An example rule definition.

```
rule ExampleRuleName
   when(DataTypeA InEvent)
   produce(DataTypeB OutEvent)
   {
      // some C++ code which can read data from InEvent
      // some C++ code which can write data to OutEvent
      if (condition) produce(OutEvent);
   };
```

where:

- ExampleRuleName is the name of this particular rule.
- DataTypeA and DataTypeB are built-in or user defined types (classes or structs).
- InEvent is the identifier of the input or triggering event whose data type is DataTypeA.
- OutEvent is the name of a possible output event whose data type is DataTypeB.

A module's rules describe (1) events of interest, (2) events it may generate, and (3) code to implement the behavior of the module (see Figure 1). When a module receives an event, it determines which of its rules have an interest and passes the event to them for processing. When triggered by an event, a rule executes a body of C++ (or C) code which may itself cause the production of new events. An event's data type information is checked when it is received by a module to ensure that the type of the event matches the type of the data expected by the rules. This prevents erroneous interpretation of the data.

Events received and produced may be of either built-in types (e.g. int, float, char), or user-defined types (i.e. classes or structs). In the simplest cases, built-in types can be used and their value assigned or read just as with normal variables of that type. If no data is to be associated with an event (i.e. it is simply used to trigger other rules), the type may be specified as "void".

In a rule definition, the when() clause lists a group of input events which must occur before the code in the rule body will be executed. At least one input event must be specified. If only one event is specified, as in Figure 1, the rule triggers every time an event with the specified identifier is received. If more than one event is specified, a logical combiner must be provided, indicating the condition under which the rule will be triggered. The possible logical combiners are *and*, *then*, and *followed by*. They may not be mixed within the same when() clause.

The *and* combiner specifies that all events in the list must be received, regardless of order, to trigger the rule, and is written as:

```
when (DataTypeA inA  and  DataTypeB inB  and  DataTypeC inC)
```

When an *and* rule is triggered, the data from all events in the list are available and represent the most recent events received. Thus, if a rule using the above

example had already received two inA events and four inC events, upon receiving an inB event the rule would be triggered and only the most recent of the inA and inC events would be available for use. The previous inA and inB events are discarded.

A *then* combiner specifies that the events in the list must be received in the specified relative order, and is written as:

```
when (DataTypeA inA  then  DataTypeB inB  then  DataTypeC inC)
```

This rule specifies that an inA event must arrive before an inB event which must arrive before an inC event. In other words, once an inA event is received, the rule can continue to receive inA or inB events but until an inB event is received, inC events are discarded.

A *followed by* combiner specifies that not only must the events be in order but they must be in exact order, and is written:

```
when (DataTypeA inA  followed by  DataTypeB inB  followed by  DataTypeC inC)
```

The only sequence of events that will trigger the rule is inA, inB, inC.

The produce() specifier lists a group of possible output events that may be generated within the body of a rule. Any event produced in the rule's body must be declared here. Zero or more events may be produced by a rule.

The main body of the rule, located within the braces, may contain any valid C++ code. This code has access to, but should not change, the data in the input events. It may (and usually must) write data into the output events. Events are treated as local variables whose data types are specified in the when() and produce() clauses.

Figure 2. A rule for processing events from a 3-D tracking device.

```
rule CloseToOrigin
   when (TrackData Position)
   produce (void NearOrigin) {
      if (Position.x < 2  &&  Position.y < 2  &&  Position.z < 2 ) {
          produce(NearOrigin);
      }
   }
```

Figure 2 shows a rule for accepting data from a 3-D tracking device and testing whether the device is within 2cm. of the origin. If it is, the rule produces an event notifying the other rules and modules of this occurrance.

A producer of an event has no complete set of expectations about what will occur as a result of producing an event, nor does it know which other rules might use the event. Thus, events should be produced whenever something happens within a module of potential interest elsewhere in the world. This kind of structure is similar in some respects to the way tuples are produced and handled within Linda, but without persistence [10].

Modules are typically constructed from one or more sets of rules. Rules are preprocessed into C++ code and then compiled and linked with the toolkit libraries and other user written C or C++ code.

3-D Graphics Support

A C++ class library is provided for building 3-D models of virtual world objects. The library enables the creation of object-oriented hierarchical models without having to learn a specific graphics library, such as GL or PHIGS. The user deals with familiar entities such as geometric shapes, and operations like translations and rotations. Objects designed using CAD or other tools may be imported and used within this library. Custom renderers may be built (as a module) for specific applications.

World Construction

Worlds are composed of interconnected modules that exchange events. One of the most important features of the toolkit is its transparent exchange of events (data) between modules. This greatly reduces the work of the application developer. Each module has one or more rules. Each rule consumes one or more events. The set of events a module consumes is the union of events consumed collectively by all the rules in the module. When two modules, say A and B, connect to each other, they exchange lists of the events they respectively consume. This enables A to determine which internally generated events (if any) to send to B and vice versa. In this way, only those events relevant to a given module are passed along, and inter-module network traffic is minimized.

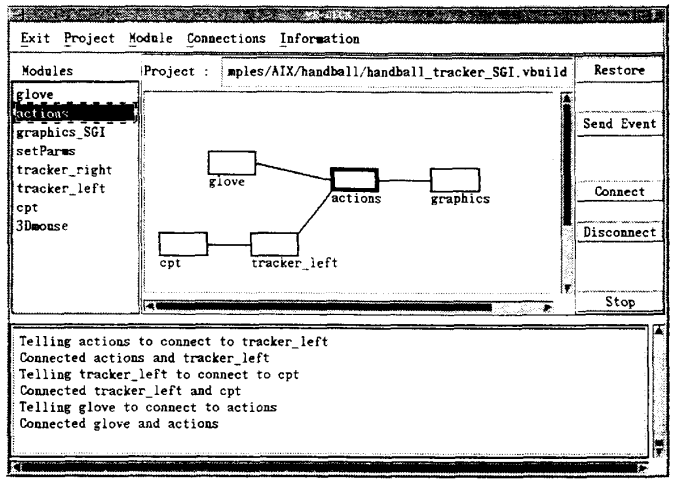
A world is constructed by choosing a collection of modules from a library and connecting them. VR-DECK includes the capability of manipulating modules through a graphical user interface environment using a tool called ***vbuild*** (see Figure 3). To the left is a list of available modules and to the right is a work area. The VR application builder selects modules from the library and drops them into the work area where they become available for construction.

Once a module is added to the work area it is ready to be started (i.e. it's process may be started on a chosen computer in the network) and connected to other modules. The starting of a module involves specifying parameters such as the path name of the executable, arguments, environment variables and location (host name) in the network. These parameters are normally specified during the construction phase and are thereafter stored for future use. Once saved, startup can happen automatically.

Within the work area, one may connect modules together, thereby selectively tailoring the topology of the module interconnections. Multi-user systems can be configured easily by providing each user with a set of appropriate modules and selectively connecting them. Optionally, modules being added to the work area can automatically connect to all other modules already there. By saving the setup and topology, the entire application can be subsequently started in one operation.

The troublesome details of starting the processs on another computer, establishing a network connection with the new process, exchanging event information, and exchanging events (data) is performed automatically by the run-time environment.

Figure 3. The ***vbuild*** graphical interface.



Extensions

The system is designed to be extended by writing new modules and adding them to the library. This simply involves writing a set of rules and compiling into a new module. For example, ***vbuild*** is itself a module. It controls and monitors other modules by sending and receiving events, and it's high level behavior is determined by a set of rules. A common set of rules is built into every module for handling the startup and interconnection protocols.[1] VR-DECK contains a library of pre-built modules for commonly used functions and devices. Included are modules for the Polhemus® trackers, the VPL DataGlove®, the Logitech® 3-D Mouse, a speech recognizer, sound generation, 3-D object-oriented graphics, and various X Window System™ control panels. A new I/O device can be added to the system by writing module rules that communicate with the device and then generate the events for other modules to receive. For example, to add a new 3-D position sensor, new rules are written that send commands to the sensor using the run-time library support for RS-232 serial I/O devices. Data returned from the device is sent to the rules which filter and reformat the data, and generate events to be received by other modules. When modules with similar function are designed to conform to a common event protocol, they can be easily interchanged without changing any other parts of the system. This permits easy and rapid substitution of system components dynamically.

Conclusion

An integrated system for designing and constructing virtual worlds has been developed. It provides a designer with an easy to use development environment while supporting distributed computing, multi-user capability, and a variety of

[1] By changing these two sets of rules, a system designer could extend or redefine the global operation of the system.

I/O devices. Virtual worlds are built as collections of modules which communicate via events. Extensive run-time support in the form of extensible C++ class libraries insulates the application designer from the low-level system details such as networking, inter-module data transport, event queuing and matching, and I/O device communication. A library of pre-defined modules is provided for commonly used functions and devices. An X Window System™ graphical user interface is provided for aggregating modules into applications. The system enables a developer to focus on the design of the application rather than on systems and integration issues. For a system developer, the toolkit is extensible, allowing integration of new devices or subsystems and applications. This toolkit brings together many of the commonly desired features of a VR development environment in an easy to use and integrated system.

References

[1] Zeltzer, D., Pieper, S. and D. Sturman,]An Integrated Graphical Simulation Platform,≠ *Proceedings of Graphics Interface '89*, pp. 266-274, 1989.

[2] Appino, Perry A., Lewis, J. Bryan, Koved, Lawrence, Ling, Daniel T., Rabenhorst, David A. and Codella, Christopher F.,]An Architecture for Virtual Worlds,≠ *Presence*, vol. 1, no. 1, 1992.

[3] C. Shaw, J. Liang, M. Green and Y. Sun,]The Decoupled Simulation Model for Virtual Reality Systems,≠ *CHI '92 Conference Proceedings*, pp. 321-328, ACM, May 1982.

[4] C. Codella, R. Jalili, L. Koved, J. B. Lewis, D. T. Ling, J. S. Lipscomb, D. A Rabenhorst, C. P. Wang, A. Norton, P. Sweeney and G. Turk,]Interactive Simulation in a Multi-Person Virtual World,≠ *CHI '92 Conference Proceedings*, ACM, May 1992.

[5] Blanchard, C., Burgess, S., Harvill, Y., Lanier, J., Lasko, A., Oberman, M. and M. Teitel,]Reality Built for Two: A Virtual Reality Tool,≠ *Proceedings of the 1990 Symposium on Interactive 3D Graphics (Snowbird, Utah)*, pp. 35-36, New York: ACM, 1990.

[6] G. Booch, *Object Oriented Design With Applications*, Redwood City, CA: Benjamin/Cummings Publishing Co., Inc., 1991.

[7] Green, M. and Beaubien, D., Minimal reality toolkit version 1.2: Programmer's manual, University of Alberta, November 1992.

[8] V. S. Sunderam,]PVM: A Framework for Parallel Distributed Computing,≠ *Concurrency: Practice & Experience*, vol. 2, no. 4, December 1990.

[9] Bricken, W., Virtual Environment Operating System: Preliminary Functional Architecture (Tech. Rep. No. HITL-M-90-2), Seattle: University of Washington, Human Interface Technology Laboratory, 1990.

[10] N. Carriero and D. Gelernter,]Linda in Context,≠ *Communications of the ACM*, vol. 32, no. 4, pp. 444-458, April 1989.